PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 7 : G06F 9/44

A1

(11) International Publication Number: **WO 00/17748**

(43) International Publication Date: 30 March 2000 (30.03.00)

(21) International Application Number: PCT/US99/21940

(22) International Filing Date: 22 September 1999 (22.09.99)

(30) Priority Data:
09/159,304 23 September 1998 (23.09.98) US

(63) Related by Continuation (CON) or Continuation-in-Part (CIP) to Earlier Application
US 09/159,304 (CON)
Filed on 23 September 1998 (23.09.98)

(71) Applicant *(for all designated States except US)*: NETCREATE SYSTEMS, INC. [US/US]; 2970 Fifth Avenue 320, San Diego, CA 92107 (US).

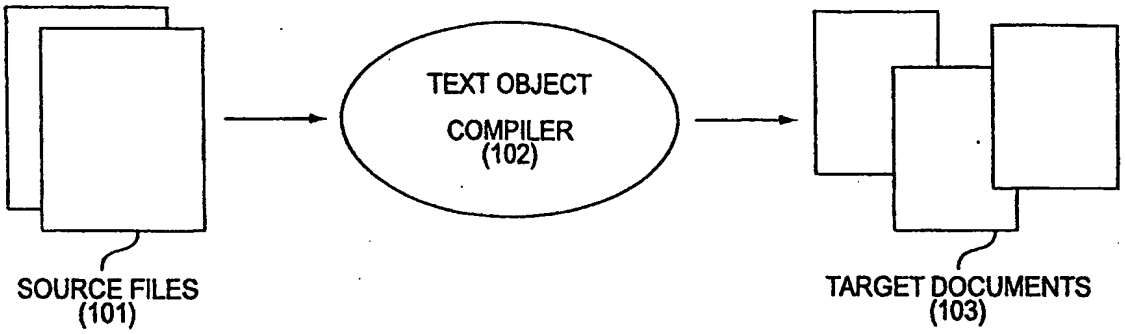
(72) Inventor; and
(75) Inventor/Applicant *(for US only)*: DOMI, Dwayne, K. [US/US]; 2970 Fifth Avenue 320, San Diego, CA 92107 (US).

(74) Agents: BURKE, John, E. et al.; Pillsbury Madison & Sutro, LLP, 1100 New York Avenue, N.W., Washington, DC 20005 (US).

(81) Designated States: AE, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CR, CU, CZ, DE, DK, DM, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SL, SZ, TZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
With international search report.
Before the expiration of the time limit for amending the claims and to be republished in the event of the receipt of amendments.

(54) Title: TEXT OBJECT COMPILATION METHOD AND SYSTEM



(57) Abstract

A Text Object Compiler and Language able to produce binary and text objects that are not machine language code. An object oriented computer language that produces target files of information in any text or binary format; files are defined by the programmer as "pages" and file locations are defined by the programmer as "targets". The compiler compiles the language to produce any variety of output, which include text formats (such as HTML, SGML, and other scripting languages) and binary formats (such as graphical pictures, binary data, or other multimedia information).

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	**ES**	Spain	**LS**	Lesotho	**SI**	Slovenia
AM	Armenia	**FI**	Finland	**LT**	Lithuania	**SK**	Slovakia
AT	Austria	**FR**	France	**LU**	Luxembourg	**SN**	Senegal
AU	Australia	**GA**	Gabon	**LV**	Latvia	**SZ**	Swaziland
AZ	Azerbaijan	**GB**	United Kingdom	**MC**	Monaco	**TD**	Chad
BA	Bosnia and Herzegovina	**GE**	Georgia	**MD**	Republic of Moldova	**TG**	Togo
BB	Barbados	**GH**	Ghana	**MG**	Madagascar	**TJ**	Tajikistan
BE	Belgium	**GN**	Guinea	**MK**	The former Yugoslav Republic of Macedonia	**TM**	Turkmenistan
BF	Burkina Faso	**GR**	Greece			**TR**	Turkey
BG	Bulgaria	**HU**	Hungary	**ML**	Mali	**TT**	Trinidad and Tobago
BJ	Benin	**IE**	Ireland	**MN**	Mongolia	**UA**	Ukraine
BR	Brazil	**IL**	Israel	**MR**	Mauritania	**UG**	Uganda
BY	Belarus	**IS**	Iceland	**MW**	Malawi	**US**	United States of America
CA	Canada	**IT**	Italy	**MX**	Mexico	**UZ**	Uzbekistan
CF	Central African Republic	**JP**	Japan	**NE**	Niger	**VN**	Viet Nam
CG	Congo	**KE**	Kenya	**NL**	Netherlands	**YU**	Yugoslavia
CH	Switzerland	**KG**	Kyrgyzstan	**NO**	Norway	**ZW**	Zimbabwe
CI	Côte d'Ivoire	**KP**	Democratic People's Republic of Korea	**NZ**	New Zealand		
CM	Cameroon			**PL**	Poland		
CN	China	**KR**	Republic of Korea	**PT**	Portugal		
CU	Cuba	**KZ**	Kazakstan	**RO**	Romania		
CZ	Czech Republic	**LC**	Saint Lucia	**RU**	Russian Federation		
DE	Germany	**LI**	Liechtenstein	**SD**	Sudan		
DK	Denmark	**LK**	Sri Lanka	**SE**	Sweden		
EE	Estonia	**LR**	Liberia	**SG**	Singapore		

TEXT OBJECT COMPILATION METHOD AND SYSTEM

BACKGROUND OF THE INVENTION

Field of the Invention

This invention relates to the field of software compilers, and more particularly relates to a method of generating files of information from one or more source files.

Description of the Related Art

In March, 1989, the European Laboratory for Particle Physics or CERN (Conseil Europeen pour la Recherche Nucleaire) developed the World-Wide-Web (WWW, or simply, "the web"), an Internet-based computer network that allows users on one computer to access information stored on other computers through a world-wide network. With an intuitive user-interface, known as a web browser, the web rapidly became a popular way of transmitting and accessing text and binary information. Since then, there has been a massive expansion in the number of World-Wide-Web sites, and the amount of information placed on the web.

Information, in the form of electronic files, documents, images, sounds and other formats, forms the basis of internet and web content, and the key to creating a useful and meaningful web-site.

To place information on the web, the information must be stored in a binary or text format in a "file." Binary documents are saved in known formats that depend upon the information being stored. For example, two-dimensional pictures are often stored in "Joint Photographic Experts Group" (JPEG) or "Graphical Image Format" (GIF) standard formats. Audio files and moving images have other formats as well, such as "WAV," "MOV," and "MPEG." For text documents, documents are stored in a HyperText Markup Language (HTML) format. The HTML format dictates the appearance and structure of a web text document, also referred to as a "web page."

Although these formats are required to create compatibility for web browsers, modifying web sites and updating information in these rigid formats is difficult and time consuming. For example, suppose every web page had a copyright notice on it. To update the copyright notice on every page, a web-site administrator would have to either change every page by hand, or use a method of global-search-and-replace. However, because of the non-uniform manner of some web-sites, a global-search-and-replace may

not work. More complicated web page changes, such as modifying small applications, known as "applets," are even more difficult. It would be much better if there was a single location or file that could be updated, and the change would be propagated to the entire web-site, or just the appropriate web pages. Very simply put, the problem of maintaining and generating large amounts of data, in any format, is difficult and highly time consuming.

Several solutions have been proposed, each has its problems.

Some web developers choose to generate web pages through a "what you see is what you get" (WYSIWYG) web-page editor. Such editors assemble web pages through a graphical interface, which makes designing pages simpler, but the results are limited because it does not solve the need to maintain the information. Using the above example, to update the copyright notice on every page, a web-site administrator would still have to edit the web pages individually, or the web-page editor program may use a method of global-search-and-replace.

Alternatively, simple pre-processor programs have been used to assemble HTML files. Such pre-processors allow web-page designers to pre-process documents and insert listed documents into a master document. For example, to include another listed document file called "foo.doc" into the master document, a web-page designer could type:

```
#include "foo.doc"
```

and the listed document would be included. While this allows fragments of common HTML code to be inserted into documents, as a web-site grows, and more pages are added to the site, the maintenance of such a system quickly becomes a logistical nightmare. Also, the fragments cannot be redefined at the point that they are included in a document. Moreover, such a system is limited strictly to text-based documents, and cannot handle binary forms of information.

U.S. Pat. No. 5,181,162, issued January 19, 1993 to Smith et al. entitled "Document management and production system," incorporated herein by reference, discloses a system of decomposing documents into logical components, which are stored as discrete "objects" in an object-oriented computational environment. The system relies on queries to a relational database which occur every time the document is printed, displayed electronically, or electronically transmitted. For a web site, which may transmit pages thousands of times per minute, this solution is a burden on the web server's computing resources. Consequently, the system would be slow, and of limited usefulness to such a high-demand environment. Similarly, the use of a relational database to deliver

pages of information on client machines has been attempted; while this provides dynamic construction of documents when they are delivered to the client machines, this solution also burdens the server's computing resources because page information would be constantly regenerated. Although caching generated pages may solve some of the computing resource problems, it creates a new problem because cached pages may be outdated.

Several related patents, U.S. Pat. No. 5,668,999, which issued September 16, 1997 to Gosling ("System and method for preverification of stack usage in bytecode program loops,"), U.S. Pat No. 5,692,047, issued to McManis ("System and method for executing verifiable programs with facility for using non-verifiable programs from trusted sources,") and U.S. Pat No. 5,706,502, issued to Foley et al. ("Internet-enabled portfolio manager system and method,"), all incorporated herein by reference, also fail to solve the problem. Collectively, these patents disclose a method and system of verifying the integrity of computer programs written in a bytecode language to run applications remotely on a client workstation. While this solution may create dynamic client-machine applications, it does not solve the problem of maintaining information in a system.

What is needed is a more flexible way of handling both binary and text information that can produce files of different file formats and still be easy to maintain.

The invention, a Text Object Compiler method, allows users to abstract information, and produce information in virtually any file format.

Almost every contemporary computer is a register-based Von Neuman computer that responds to a machine language. These machine languages include instructions which operate on the contents of registers. Originally, computer software instructions were organized in terms of machine language operations. As computers became more complex, programming in machine language became difficult and increasingly cumbersome. Consequently, computer scientists abstracted machine instructions, creating higher-level languages, known as source languages, structured in terms of expressions and procedures. As software evolved, two strategies for converting source languages into instructions in machine language developed, interpreters and compilers.

An interpreter, written in the native machine language, configures the computer to execute programs written in one of the source languages. The primitive operators or commands of the source language are implemented as a library of subroutines written in the native machine language of the given machine. Interpreters read the source language,

one line at a time, and then perform the specified operation. A program to be interpreted, the source program, is represented as a data structure. The interpreter traverses this data structure, analyzing the source program. As it does so, it simulates the intended behavior of the source program by calling appropriate primitive operators from the library.

Instead of analyzing and translating the source program into machine language during execution, it is possible to perform these tasks before execution, enabling more efficient program execution. This alternate method of converting source languages into instructions is called compilation. The program that does the analysis of the source program and reduces the source program to machine language is called a compiler. As shown in FIG. 1, a conventional (i.e., prior art) compiler **2** for a given source language and machine translates computer source code **1** (i.e., a program written in a high level "computer language") into an object code **3**, a program written in the computer's native language, referred to in the art as "machine language."

Illustrated by FIG. 2, a conventional compiler **2** is composed of a lexical analyzer **10**, a parser **20**, and code generator **30**. A lexical analyzer **10** takes computer source code **1** and divides the code into lexical tokens. Such lexical tokens can be based on instructions or other keywords in the relevant high level computer language. A parser **20** takes the tokens and groups them together logically based on the relationships established by the source language and the computer source code **1**. Lastly, a code generator **30** takes the relationships established by the parser **20** and translates them into an executable computer object code **3** in computer machine language.

Conventional compilers are well known in the prior art, such as U.S. Pat. No. 5,560,015 ("Compiler and method of compilation" issued to Onodera on September 24, 1996), U.S. Pat. No. 5,442,792 ("Expert system compilation method" issued to Chun on August 15, 1995), and U.S. Pat. No. 5,768,592 ("Method and apparatus for managing profile data" issued to Chang on June 16, 1998) all incorporated herein by reference.

Conventional interpreters and compilers convert high-level computer source code into object code to be executed on a computer. In effect, the interpreter and the compiler allow computer programmers to write computer programs at a higher level of abstraction, and generate object code.

SUMMARY OF THE INVENTION

The invention, a Text Object Compiler (TOC) method and system, applies this same level of abstraction to information as a conventional compiler applies to computer

programs. A user designs abstract source code which is compiled into a file or a plurality of files, which is not object machine language code. Instead, the TOC produces information in virtually any information format, as text or binary files. The TOC reads one or more source files written as a Text Object Language (TOL) in ASCII text and processes the source files into one or more output files in any document format. The compilation process reorganizes the information in the source files into output document formats, and may contain compile-time utility commands to facilitate the document generation process.

In the first embodiment, a lexical analyzer tokenizes a source input written in TOL regular expressions to produce a token representing the source input. Such TOL expressions may contain variables, functions, and classes. A parser determines the relationships between the tokens so that a page generator can evaluate the tokens and the relationships between the tokens to generate an output. The resulting output may be written as a file at a target location specified by the source input.

In another aspect of the invention, the source input is lexically analyzed to produce tokens representing regular expressions of the source input. The regular expressions are written in the Text Object Language and may include variables, functions and classes. The regular expressions are parsed to determine the relationship between the tokens. For instance, any class relationship between tokens is determined within this step. Finally the tokens and relationships are evaluated to generate a non-executable output. The non-executable output may then be written to a file. The file location may be specified by the original source input as a specific target location.

In the preferred embodiment of the present invention, the source input, written regular expressions of a Text Object Language, is initially read. The regular expressions contain page definitions used to determine the output of the process, and may additionally contain variables, functions, target locations and object-oriented classes. Once read, the source input is lexically analyzed to produce token representations of the regular expressions, which includes tokens generated from the page definitions. The tokens are parsed to determine the relationship between the tokens, and the resulting relationship is constructed in computer memory. Each variable and function is evaluated and their value is determined. The determined value replace their corresponding variable or function token in computer memory. The non-executable file based on the computer memory representation of the page tokens is then written to a file at a target location specified

within the source input.

BRIEF DESCRIPTION OF THE DRAWINGS

The foregoing and other objects, features, and advantages of the present invention will be better understood in view of the following detailed description made in conjunction with the accompanying drawing in which:

FIG. 1 diagrams an overview of the conventional compiler process (prior art);

FIG. 2 illustrates the basic components of a conventional compiler (prior art);

FIG. 3 diagrams an overview of the Text Object Compiler (TOC) process;

FIG. 4 illustrates the basic components of a Text Object Compiler;

FIG. 5 is a flowchart of the lexical analysis and parsing subprocesses used by the Text Object Compiler.

FIG. 6 is an inheritance diagram of the classes of the Text Object Language source file listed in Table 3;

FIG. 7 is an inheritance diagram showing the relationship of class variables and the classes;.

FIG. 8 is an inheritance diagram showing the relationship of class functions and the classes;

FIG. 9 is an inheritance diagram consolidating the classes with their functions and variables;

FIG. 10 is an inheritance diagram showing the relationship of the defined pages and the classes;.

FIG. 11 diagrams the baseclass information used to generate the target document "first.txt";

FIG. 12 diagrams the baseclass, and myclass information used to generate the target document "second.txt";

FIG. 13 diagrams the baseclass, myclass, and newclass information used to generate the target document "third.txt"; and

FIG. 14 is a flowchart detailing the page generation subprocess.

DETAILED DESCRIPTION OF THE INVENTION

The Text Object Compiler uses a variety of existing programming and compiler methods that are commonly used in programming languages to create executable files. The unique aspect of TOC is that it is applied to creating non-executable files, which are

referred to as "target documents." As shown in FIG. 3, source files **101**, written in a Text Object Language (TOL) are compiled by the TOC **102** to produce target documents **103** as output. The target document locations are definable by the programmer, and the location is referred to as a "target location." Target documents **103** can be any type of format, and can even produce other source files used by other compilers or programs. The TOC **102** actually knows nothing about the format of the target documents **103**; the target document format is solely up to the programmer.

Reference will now be made in detail to the preferred embodiments of the invention, examples of which are illustrated in the accompanying drawings. While the invention will be described in conjunction with the preferred embodiments, it will be understood that they are not intended to limit the invention to those embodiments. On the contrary, the invention is intended to cover alternatives, modifications and equivalents, which may be included within the spirit and scope of the invention as defined by the appended claims.

The Text Object Language (TOL)

In the preferred embodiment, the source file or files are written in a Text Object Language, which when compiled or interpreted will result in at least one target file. A listing of some of the TOL operators is provided in Table 1.

Operator	Description
`:=`	The equality operator tells the compiler to define the keyword on the left-hand-side of the operator as the value on the right-hand-side of the operator. For example, the usage "`length := 5`" would define the variable "`length`" with the value of 5.
`//`	This operator indicates to the compiler the presence of a comment-line. The compiler ignores the remainder of the line.
`#include := filename`	Imports a source file named "filename" for compilation.
`class classname := baseclass`	Defines a page class. *Classname* must be specified. *Baseclass*, if omitted, is assumed to be the base class for TOL. A class inherits all of the variables and functions of its base class. Classes can be *public*, *private*, or *protected*. A class can declare other classes as *friends*; a friend class is given public access to all of the declaring class' variables and functions. Within a class, variables or functions can be declared friends.

`Cleartargets := directory`	Clears all targets previously inline.

Debug := [*on/off*]	Enables/Disables debug while compiling. The default is to stop the debugger from source code.
func(*classname*) := *name* (var1, … , varN = value) **endfunc**	Begins a new function. If classname is omitted, then the class is assumed to be the base class. The function name, *name*, must be unique for a class. The function can contain zero or more variables, *var1*, *var2*, etc. Functions can be overloaded, with several functions of the same name, but with different number of variables; overloaded functions must contain a unique number of variables. Functions can be *public*, *private*, or *protected*. Functions can be made virtual, forcing them to be defined in a derived class. Once a function is declared virtual, all derived class instances of the function are virtual. Functions can contain a default value, for example, *var1=default*. *Endfunc* ends a function section.
lfcr := [*on/off*]	Enables/Disables output file wrapping.
page(*classname*) := filename **endpage**	Begins a new page section. If *classname* is omitted, then the class is assumed to be the base class. The compiler builds an output page for each page/endpage section. Within this section, all variables and functions are resolved. *Filename* is the fully qualified location of the output file. *Endpage* ends a page section.
targets (*classname*) **endtargets**	Targets allow pages output be directed to different or multiple locations. Example targets include any media or memory storage device, such as disk drives, networked drives, FTP locations, memory cards. *Endtargets* ends a target section.
vars (*classname*) **endvars**	Begins a new variables section. If *classname* is omitted, then the class is assumed to be the base class. Variables can be *public*, *private*, or *protected*. Variables can be made virtual, forcing them to be defined in a derived class. Once a variable is declared virtual, all derived class instances of the variable are virtual.

Table 1. TOL Operators

The TOL is similar to other programming languages, in that it has variables, classes, functions and subroutines, but uses a language syntax recognized only by the TOC **102**.

In addition, to the TOL operators, a number of compile-time utility commands exist to facilitate the document generation process

Utility Command	Description
beep := *length*	Causes the system to sound beep with duration of length in milliseconds. The default duration is 100 milliseconds.
chdir := *path*	Changes the current directory to *path*. Any reference to a path or filename that does not include a drive letter will default to the current drive. "chdir" is executed by the compiler inline.
Chdrive := *driveletter*	Changes the current drive. Any reference to a path or filename that does not include a drive letter will assume the current drive. "chdrive" is executed by the compiler inline.
copy := *source, destination*	Copies source to destination when the compiler reaches this inline command.
exec := *program*	Runs the specified application in program.
kill := *filespec*	Deletes the files specified in *filespec*.
md := *path*	Creates the directory *path*.
rd := *path*	Removes the directory *path*.

Table 2. TOL Compile-Time Utility Commands

An exemplary source file 101 written in the Text Object Language of Table 1 can be seen in Table 3.

```
//define the class structure and relationships
class myclass := baseclass
class newclass := myclass

//variable definitions

//baseclass variables
vars
        title := This is my document title
endvars

//variables for myclass pages only
vars(myclass)
        title := This is the document title for myclass
endvars

//functions

//baseclass function example
func:=myfunc (var1, var2)
        <H1>var1</H1>
        <H2>var2</H2>
endfunc

//function for newclass pages only
func(newclass):=myfunc(var1, var2)
        <center>
                var1<br>
                var2
        </center>
endfunc

//target documents
page:=first.txt
        myfunc(title, This is a base class example)
endpage

page(myclass):=second.txt
        myfunc(title, This is a myclass example)
endfunc

page(newclass):=third.txt
        myfunc(title, This is a newclass example)
endpage

//target locations
targets
        Local Drive := c:\web
        LAN Drive := n:\web
        Live Site := ftp://www.netcreate.com/web/html
Endtargets
```

Table 3. Example Source File written in the Text Object Language ("mysource.txt")

As illustrated in the example Text Object code source file **101** of Table 3, there are five primary components of TOL: classes, variables, functions, pages, and targets. Although the concepts of classes, variables, and functions exist in prior art computer languages, the additional concepts of pages and targets exist in the Text Object Language.

TOL classes are similar to and share many of the elements of common Object Oriented Programming (OOP) classes. Classes allow a document programmer to organize document sections into objects that can be reused throughout the source files and applies to any of the target files. Specifying classes is optional, since a default class, or "base class," is always assumed. A "derived" or "child" class inherits all of the variables and functions of its base or "parent" class.

Classes can be *public*, *private*, or *protected*. Public classes allow their functions and variables to be redefined by other classes. By default, all classes are public. Private classes allow their functions and variables to be redefined only by other member or friend classes. Protected classes allow its variables and function to be used only by member functions, friends of the class in which it is declared, and by member functions and friends of classes derived from the protected class. In addition, a class can declare other classes as *friends*; a friend class is given public access to all of the declaring class' variables and functions.

Variables allow the source file programmer to represent elements of a target document by reference, and use the reference to create sections or target documents rather than using the actual data. Like classes, in the preferred embodiment, variables can be can be *public*, *private*, or *protected*. Variables can also be made virtual, forcing them to be defined in a derived class; however, once a variable is declared virtual, all inherited class instances of the variable are virtual. Note that no virtual variables of the class may exist within the program until the virtual variable is defined by the derived (child) class.

A function is a convenient way to encapsulate some computation, which can then be used without worrying about its implementation. Functions allow programmers a conceptual way to abstract a recurring procedure without worrying about the details. Functions are similar to typical programming subroutines. Like classes and variables, functions can be *public*, *private*, or *protected*.

Function name overloading allows multiple function instances that provide a common operation on different argument types to share a common name. Functions can be overloaded, with several functions sharing the same name, but each having a different

number of variables. Each overloaded function must have a unique number of variables, which allows the compiler to distinguish between each instance of the overloaded function. Functions can be made virtual, forcing them to be defined in a derived (child) class. Once a function is declared virtual, all derived class instances of the function are virtual. A virtual baseclass function is also virtual in the derived class if inherited by the derived class; such a function is treated as an abstract class, and no objects of the class may exist within the program until the function is defined by a derived class.

Pages are unique to the Text Object Language; page parameters instruct the Text Object Compiler **102** how to combine or parse source files **101** into the actual individual target documents **103**. A page defines the starting and ending point of a resulting target document **103**, and the contents of the target document **103**.

Targets are also unique to the Text Object Language. Target parameters define the target location; once defined, the target parameters instruct a Text Object Compiler **102** on where to place the target documents. This location is referred to as the "target location." The target location may be local to the computer running the TOC **102**, or at a remote location that can be accessed over a computer network by the computer. If the source files **101** define multiple target locations with the target parameter, the TOC **102** will produce identical target documents **103** at each target location. Multiple targets are useful for creating experimental output, creating backup files for redundancy purposes, and updating main/production server files. For example, a programmer may define two targets to create a primary web-site and its "mirror" web-site at an alternate location. If target parameters are omitted from the Text Object code **101**, the target documents **103** will be created in a default local location.

The Text Object Compiler (TOC)

The Text Object Compiler **102** performs the compilation of the text object language source files **101**, resulting in target documents **103** defined by the pages parameter as output at a target location defined by the targets parameter. As discussed, target documents **103** may be in any format; note that this distinguishes the TOC **102** from prior art software compilers that which only produce object machine language code, i.e. executable files. Note however, that programmers define the output format of the files with their source program code **101**.

Attention will now be given to the TOC structure and method.

The TOC **102** is similar in structure to a conventional compiler. Like a

conventional compiler, the TOC contains a lexical analyzer **10** and a parser **20**. However unlike the TOC, a conventional compiler, shown in FIG. 2, feeds parser output into a computer code generator **3**, to generate executable computer object code **3**. As illustrated in FIG. 4, in a Text Object Compiler **102**, the parser **20** output is presented to a page generator **200** to produce the target documents **103** as output.

The TOC lexical analyzer **10** examines expressions in a similar fashion to a conventional compiler lexical analyzer. This division into units, known as "tokens," is a process known in the art as "lexical analysis." Essentially, the lexical analyzer looks for regular expressions. A regular expression is a pattern description using the computer language. The lexical analyzer performs as many regular expression matches as possible, and attempts to classify the text of the entire source file into tokens. In the Text Object Language, the expressions may include variable names, function names, class names, target locations, page definitions, constants, strings, operators, punctuation, and so forth. For example, when compiling the source file in Table 3, the compiler initially classifies each instance of a known operator (as listed in Table 1) as a known token. However, if the word or expression is unknown to the compiler, it too is still tokenized, but its value or relationship must still be determined by the parser.

As the input is divided into tokens, the compiler must establish the relationship between the tokens. The Text Object Compiler needs to find the expressions, statements, declarations, blocks, functions/procedures, class structures, and pages in the program, a process known as "parsing." The list of rules that define the relationships that the compiler understands is called grammar. The grammar of an exemplary Text Object Language is shown above in Table 1.

The Text Object Compilation process is best explained by example. An existing source file **101**, such as the example in Table 3 is written in the Text Object Language. The compiler reads the source file, as illustrated in step **250** of FIG 5. In its process of compiling the source code, a compiler performs two tasks over and over: a.) dividing the input source code into meaningful units (step **260**), and b.) discovering the relationship between the units (step **270**). These two processes are respectively called "lexical analysis" (step **260**) and "parsing" (step **270**). If the parser cannot determine the relationship of the token, it next determines whether the end of the source file has been reached, step **280**. If the end of the source file has been reached (step **280**), the undetermined tokens are an error in either syntax or usage, and an error is reported, step

282. If the end of the source file has not been read, the compiler loops back to step 250, and reads the source file. Similarly, if the parsing of step 270 is successful, and the entire file has not been read, as determined by step 284, the compiler continues to read more lines of the source file, step 250.

An example of the lexical analysis and parsing steps are as follows. The compiler initially reads the first line of Table 3, step 250. Each word is tokenized, and matched against a known set of regular expressions, such as the TOL Operators. The first known operator, the comment operator ("//") is identified, step 260. As defined by the implementation of this TOL grammar, the remainder of the line is determined to be a comment, and the compiler ignores the remainder of the line, step 270. Since the end of the source file has not been reached, as determined by step 284, the compilation process continues, and the compiler reads the next line of the source file, step 250.

The lexical analyzer notes the presence of four tokens on the second line, the words "class," "myclass" ":=" and "baseclass." Step 260. Two of these tokens, "class" and the equality operator (":=") are identified as operators, and a third token, "baseclass" is identified as the "baseclass" keyword, which defines the TOL base class. The token "myclass" is initially unknown by the lexical analyzer. The token information is forwarded to the parser, which realizes that the source file defines a child class "myclass" which descends from the TOL baseclass, step 270. The parser constructs a memory table, memory tree, or equivalent memory structure to categorize the class structure. The process is repeated with the next line, resulting in a class inheritance relationship depicted by FIG. 6. Class myclass 310 is derived from the base class 300, and class newclass 320 is a "child" class derived from the "parent" class myclass 310. The memory tree is expanded to reflect the newclass class.

The compiler processes the next several lines of Table 3, which consist of variable definitions for the variable "title." The variables are parsed and stored in the memory tree, linked to their appropriate class definition, as shown by FIG 7. The baseclass 300 is associated with a variable "title" 301. Similarly, myclass 310 is associated with a different definition for another variable called "title" 311.

FIG 8. illustrates the relationships of the functions declared with their defined classes. The code in Table 3 defines a "myfunc" function that is different for the baseclass 300 and newclass 320; consequently, a myfunc 302 is associated with the baseclass 300, and a different myfunc 322 function is associated with newclass 320.

FIG 9. consolidates the inheritance diagrams with their related variables and functions. Baseclass **300** has both a variable, title **301**, and a function, myfunc **302**. The class myclass **310** also has a variable, title **311**, and since it does not have a definition for myfunc, it inherits the function definition for myfunc **312** from the baseclass **300** definition of myfunc **302**. Similarly, the class newclass **320** does not have a value for the "title" variable, and thus inherits its definition for title **321** from the myclass title definition **311**. Newclass **320** does have its own definition for the function myfunc **322**, and this is also reflected in the inheritance diagram.

The lexical analysis and parsing process is repeated for both the target document and target location sections of the code. As shown in FIG. 10, the target document "first.txt" **400** is of the baseclass **300**, "second.txt" **410** is of class myclass **310**, and "third.txt" **420** is of class newclass **320**. Each of the three target documents consist of a single function call to the appropriate class function "myfunc."

Once all the source files have been tokenized and parsed to known values, the established token and relationship information is passed to the compiler page generator, step **286**.

As shown previously in FIG. 4, the compiler page generator **200** creates each target document based on the relationships and tokens forwarded from the parser **20**, replacing variables with their appropriate values, evaluating function calls, and substituting the resulting information into the page table shown in FIG. 10.

The page generator sub-process is elaborated in FIG. 14. The token relationship information is passed to the compiler page generator, step **286**. For each page class, the variables are replaced with their respective definitions, step **288**. In a simple embodiment, this can merely be the substitution of the value into each memory table location where the variable appears. Each function call for every page class is then evaluated, step **290**. The existence of each named target location is verified; if the location, such as a directory, does not exist, it may be created at this time by the compiler, step **292**. Each page, corresponding to a target document, is then written at each target location, step **294**. Lastly, the write is verified by the compiler, step **296**.

For example, as illustrated in FIG. 11, the output for "first.txt" **400** is generated by noting the appropriate class, baseclass **300**, which defines the functions and variables used in generating the page. Table 3 defines "first.txt" as a page generated by a function call to "myfunc" using the "title" variable and "This is a base class example" as the input.

Since "first.txt" is of class baseclass, the definitions for "title" and "myfunc" are taken directly from the baseclass. The results are shown in Table 4.

```
<H1>This is my document title</H1>
<H2>This is a base class example</H2>
```

Table 4. Compiled output for "first.txt"

FIG. 12 continues the compilation for "second.txt" **410**, which is of class "myclass" **310**. The definitions for "title" is taken directly from class myclass **310**. The definitions for "myfunc" would normally also be taken from class myclass **310**. However, since "myfunc" is not defined for myclass **310**, the myfunc function definition for myclass' parent class, baseclass **300**, is used. The compiled results for "second.txt" **410** are shown in Table 5.

```
<H1>This is the document title for myclass</H1>
<H2>This is a myclass example</H2>
```

Table 5. Compiled output for "second.txt"

FIG. 13 continues the compilation for "third.txt" **420**, which is of class "newclass" **320**. The definitions for "title" and "myfunc" are normally taken directly from class newclass **310**. However, since the "title" variable is not defined for newclass **320**, the "title" variable definition for newclass' parent class, myclass **310**, is used. Since "myfunc" is defined for the class newclass **310**, the newclass "myfunc" definition is used. The compiled results for "third.txt" **420** are shown in Table 6.

```
<center>
        This is the document title for myclass<br>
        This is a newclass example
</center>
```

Table 6. Compiled output for "third.txt"

Once the compiler generates each page in memory, each page is written as a target document at each target location. The compiler may optionally create previously non-existing target locations, and verify the writing at the target locations; in its most preferred embodiment, the TOC performs both actions, reporting a warning message if a target location is not created, or an error message if a problem in writing the target document occurs.

CLAIMS

We Claim:

1. A computer comprising:

a. a lexical analyzer that tokenizes a source input to produce tokens representing regular expressions of the source input;

b. a parser that determines relationships between the tokens;

c. a page generator that evaluates the tokens and the relationships between the tokens to generate an output, the output being non-executable.

2. A computer of claim 1 wherein the page generator writes the output as a file.

3. A computer of claim 2 wherein the page generator writes the file at a target location specified within the source input.

4. A computer of claim 1 wherein the regular expressions include variables.

5. A computer of claim 1 wherein the regular expressions include functions.

6. A computer of claim 1 wherein the regular expressions include classes.

7. A computer of claim 1 wherein the regular expressions include variables, functions, and classes.

8. A method of operating a computer system to compile a Text Object Language comprising:

lexically analyzing a source input to produce tokens representing regular expressions of the source input;

parsing the tokens to determine relationships between the tokens;

evaluating the tokens and the relationships between the tokens to generate a non-executable output.

9. A method of claim 8 further comprising:

writing the non-executable output to a file.

10. A method of claim 9 wherein the file is written at a target location specified within the source file.

11. A method of claim 8 wherein the regular expressions include variables.

12. A method of claim 8 wherein the regular expressions include functions.

13. A method of claim 8 wherein the regular expressions include classes.

14. A method of claim 8 wherein the regular expressions include variables, functions, and classes.

15. A method of operating a computer system to compile a Text Object Language comprising:

reading a source input containing regular expressions of the Text Object Language, wherein the regular expressions of the text object language include variables, functions, and page definitions;

lexically analyzing the source input to produce tokens of the regular expressions, wherein the tokens include page tokens;

parsing the tokens to determine relationships between the tokens;

constructing a representation of the tokens and their relationships in computer memory;

evaluating the tokens that represent variables and functions to determine their evaluated values;

replacing the computer memory representation of the variable tokens and the function tokens with their evaluated values;

writing non-executable files based on the computer memory representation of the page tokens.

16. A method of claim 15 wherein the non-executable files are written in a target location specified by the source input.

17. A method of claim 15 wherein the regular expressions include classes.

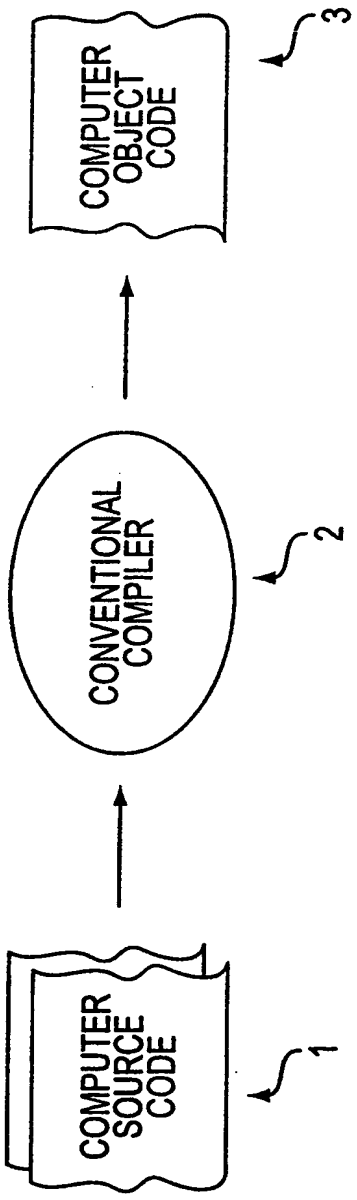


FIG. 1
(PRIOR ART)

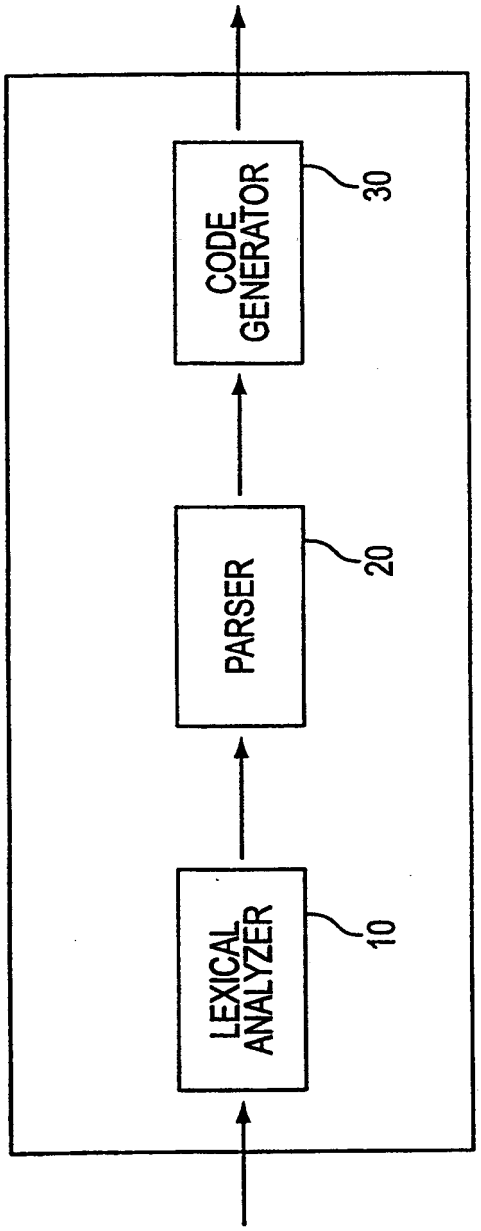


FIG. 2
(PRIOR ART)

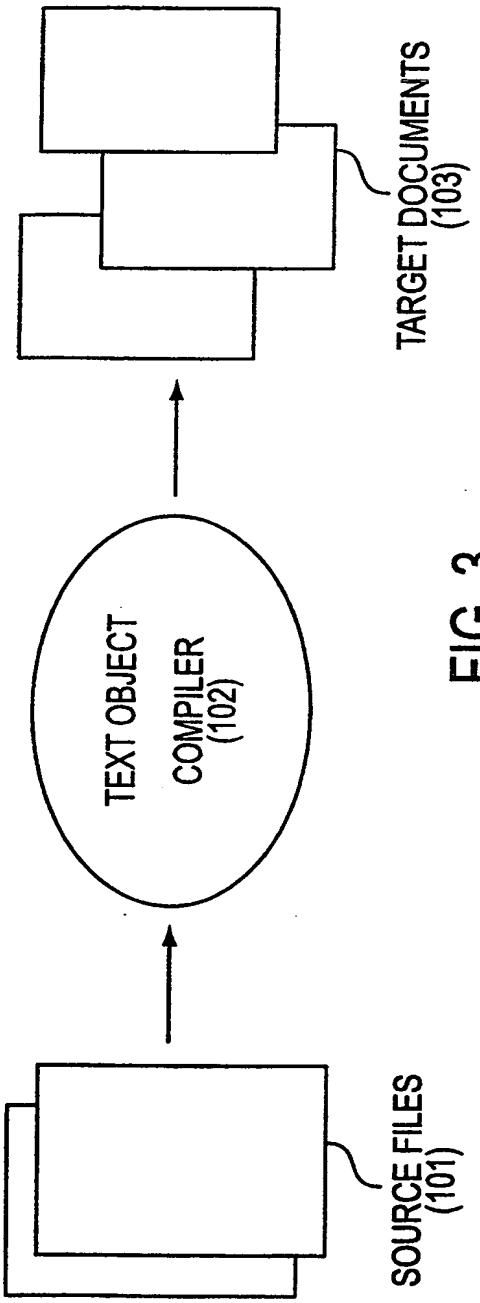

TARGET DOCUMENTS
(103)
TEXT OBJECT
COMPILER
(102)
SOURCE FILES
(101)

FIG. 3

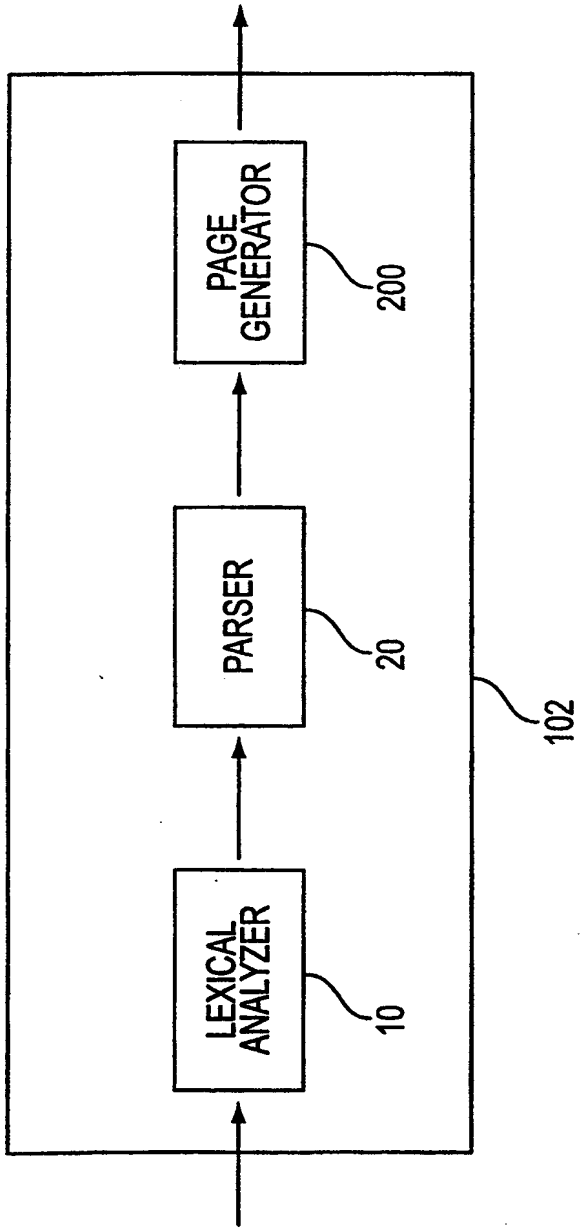


FIG. 4

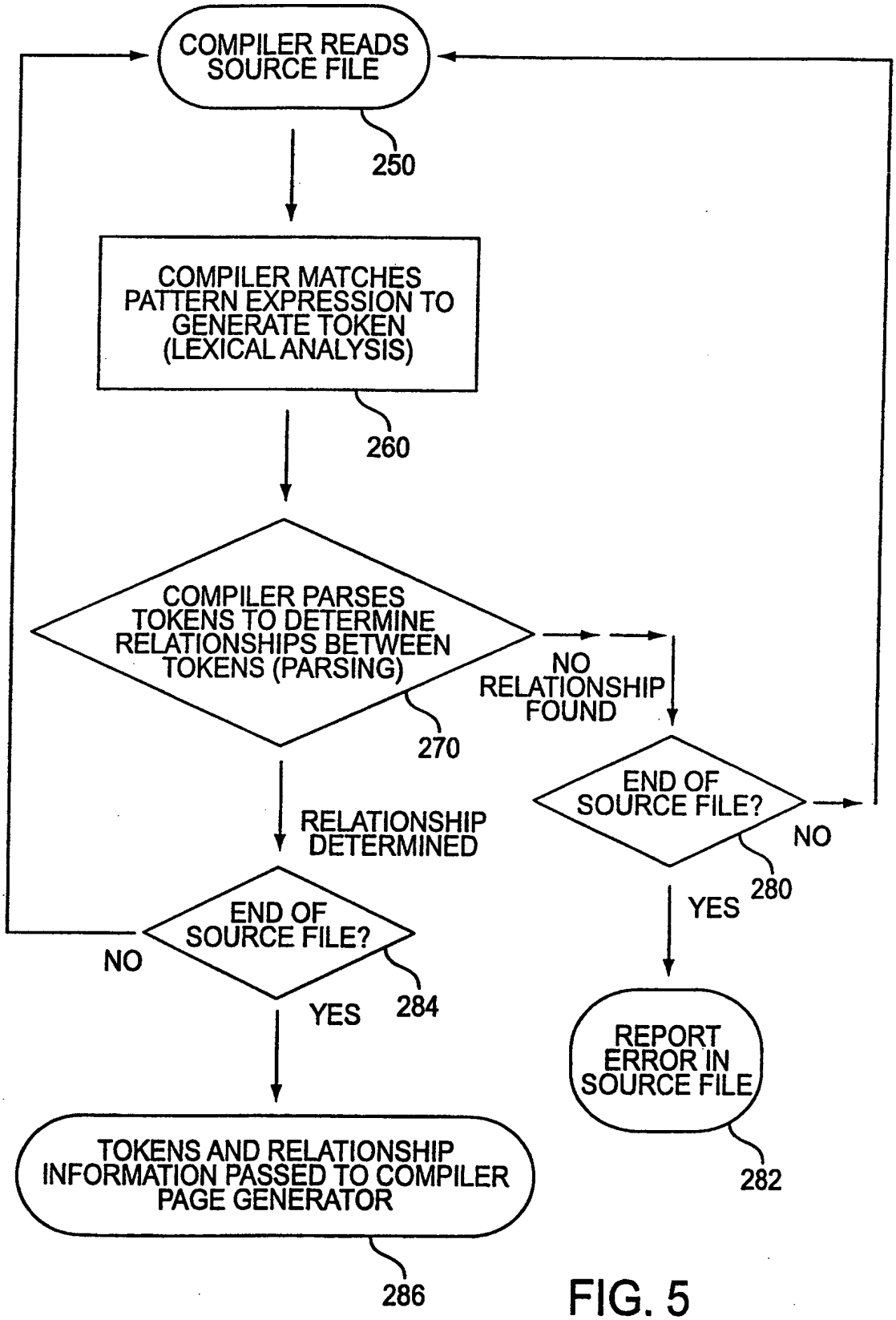

COMPILER READS SOURCE FILE
250
COMPILER MATCHES PATTERN EXPRESSION TO GENERATE TOKEN (LEXICAL ANALYSIS)
260
COMPILER PARSES TOKENS TO DETERMINE RELATIONSHIPS BETWEEN TOKENS (PARSING)
270
NO RELATIONSHIP FOUND
END OF SOURCE FILE?
NO
280
YES
REPORT ERROR IN SOURCE FILE
282
RELATIONSHIP DETERMINED
END OF SOURCE FILE?
NO
284
YES
TOKENS AND RELATIONSHIP INFORMATION PASSED TO COMPILER PAGE GENERATOR
286

FIG. 5

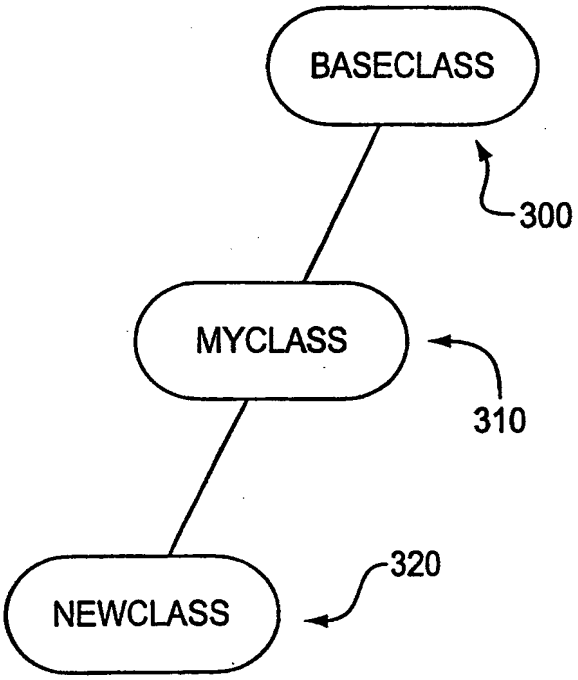


FIG. 6

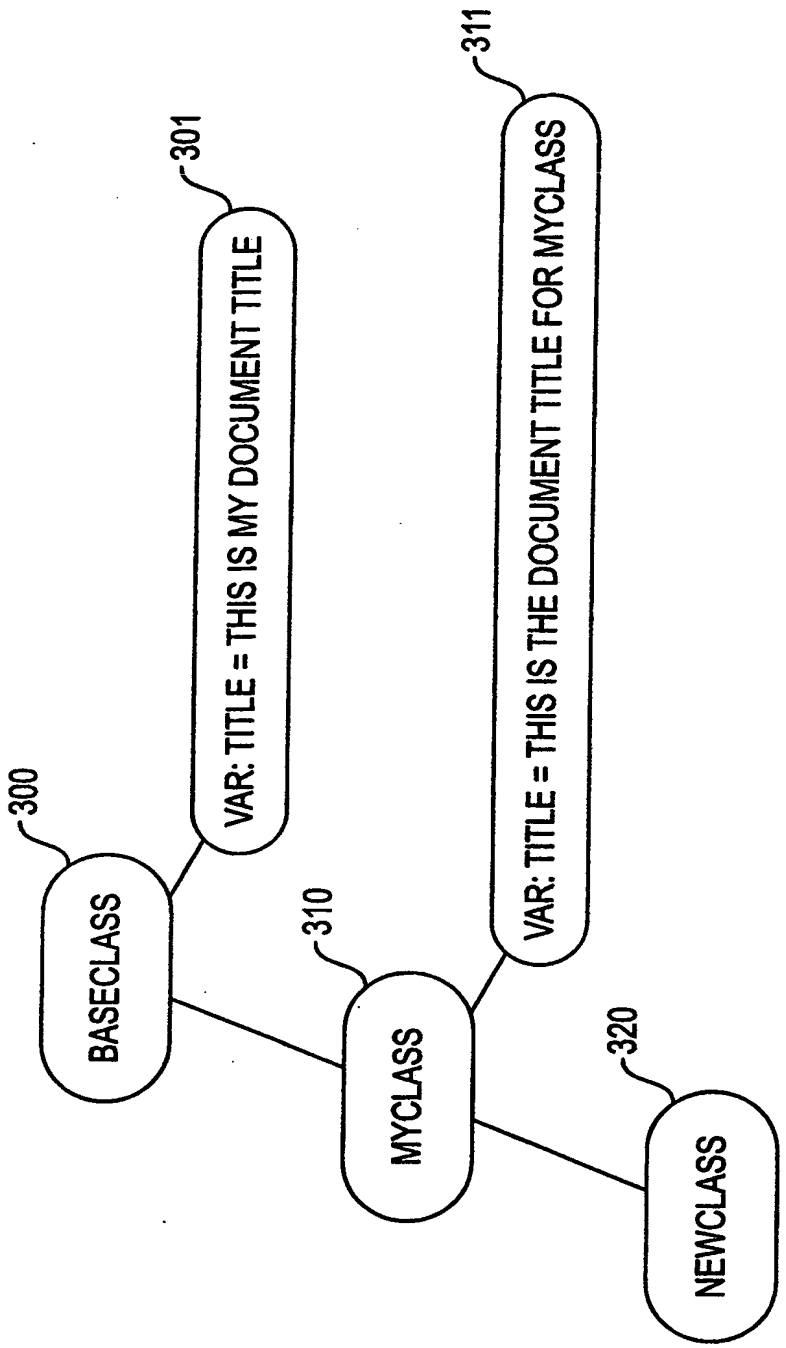

300
BASECLASS
301
VAR: TITLE = THIS IS MY DOCUMENT TITLE
310
MYCLASS
311
VAR: TITLE = THIS IS THE DOCUMENT TITLE FOR MYCLASS
320
NEWCLASS

FIG. 7

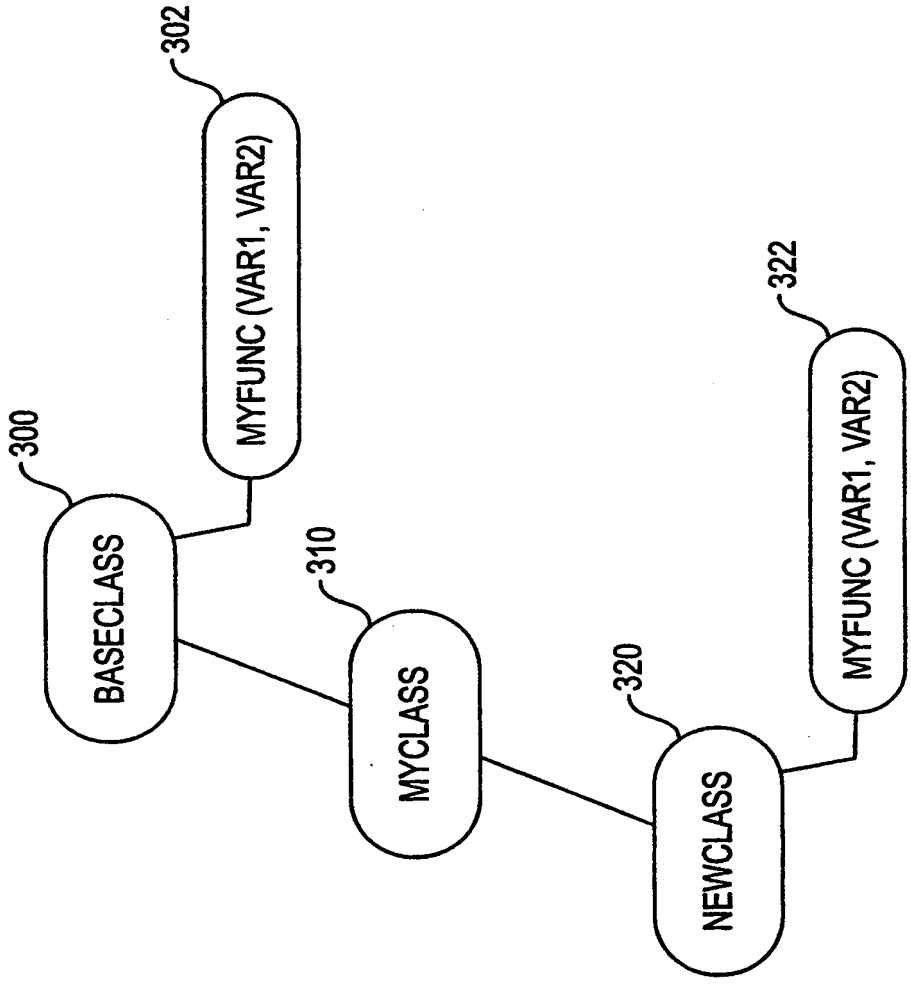

300
BASECLASS
302
MYFUNC (VAR1, VAR2)
310
MYCLASS
320
NEWCLASS
322
MYFUNC (VAR1, VAR2)

FIG. 8

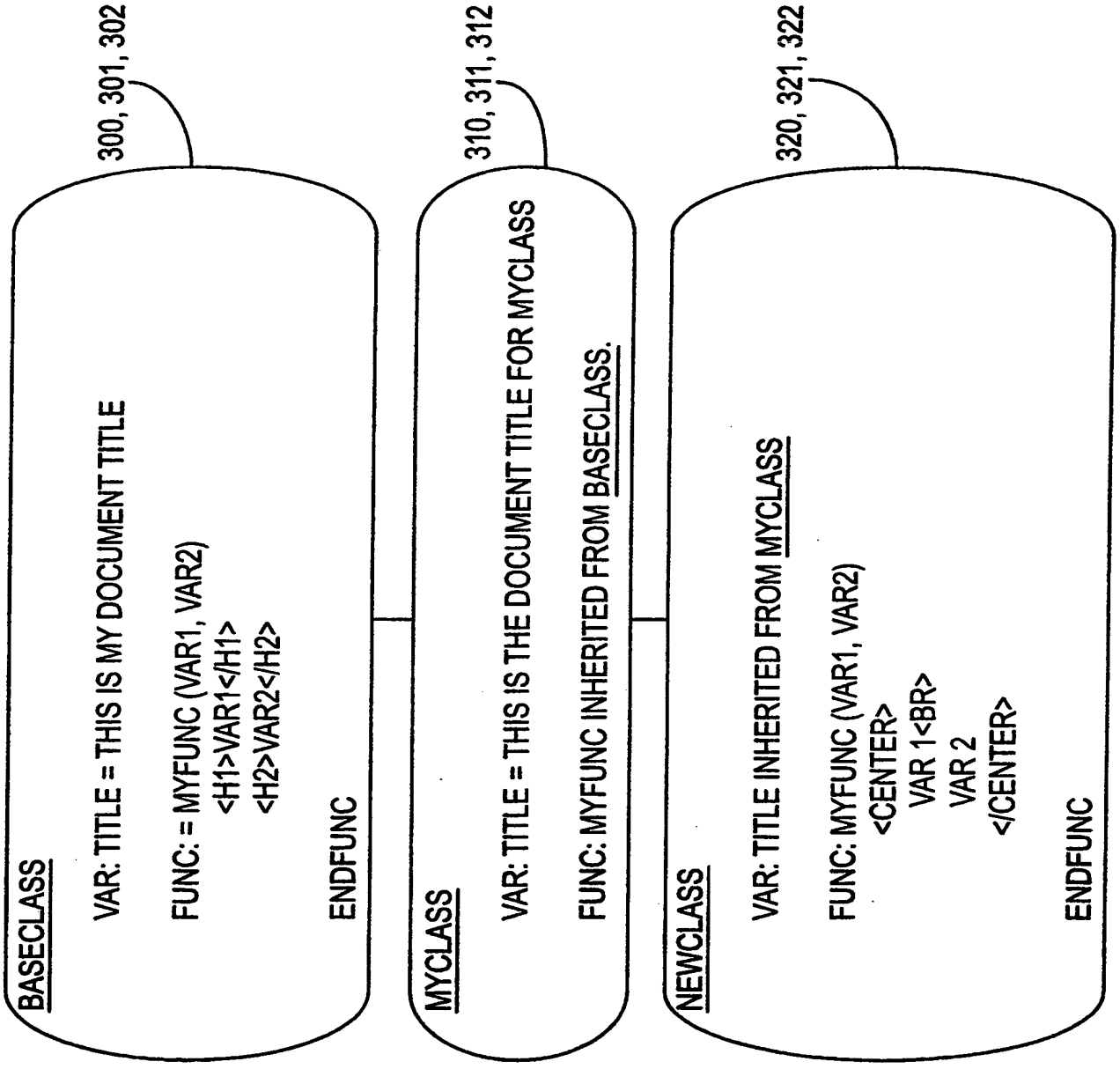

300, 301, 302
BASECLASS
VAR: TITLE = THIS IS MY DOCUMENT TITLE
FUNC: = MYFUNC (VAR1, VAR2)
<H1>VAR1</H1>
<H2>VAR2</H2>
ENDFUNC
310, 311, 312
MYCLASS
VAR: TITLE = THIS IS THE DOCUMENT TITLE FOR MYCLASS
FUNC: MYFUNC INHERITED FROM BASECLASS.
320, 321, 322
NEWCLASS
VAR: TITLE INHERITED FROM MYCLASS
FUNC: MYFUNC (VAR1, VAR2)
<CENTER>
VAR 1
VAR 2
</CENTER>
ENDFUNC

FIG. 9

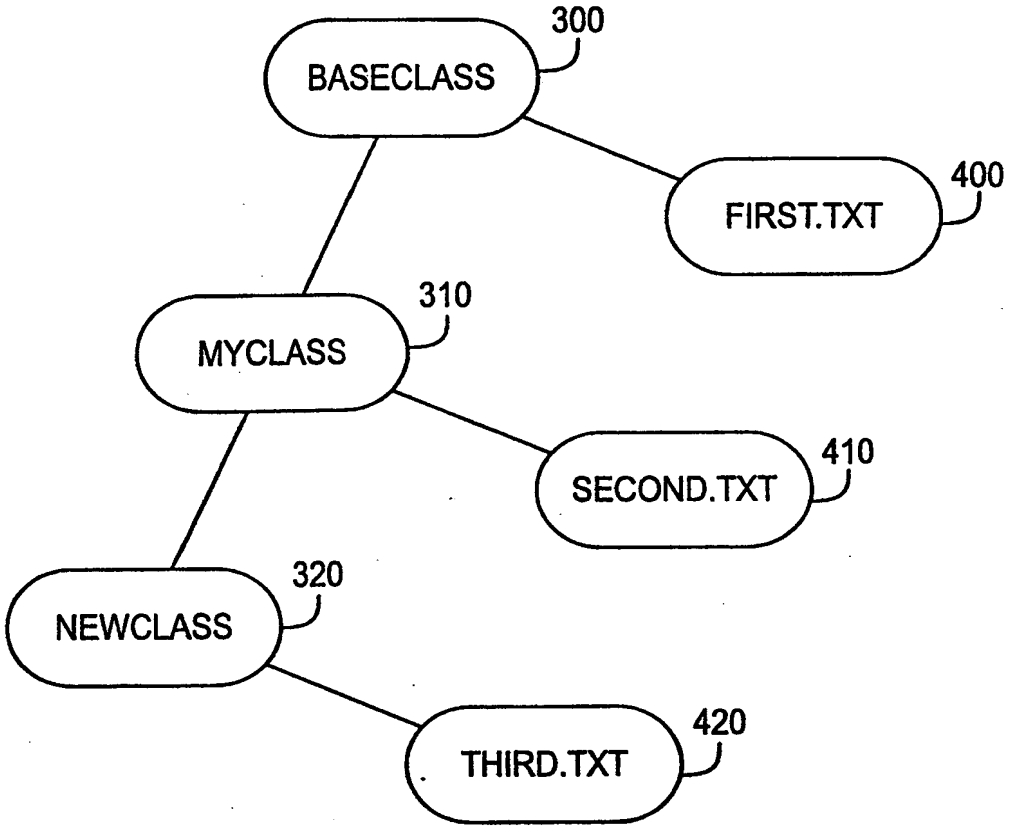

300
BASECLASS
400
FIRST.TXT
310
MYCLASS
410
SECOND.TXT
320
NEWCLASS
420
THIRD.TXT

FIG. 10

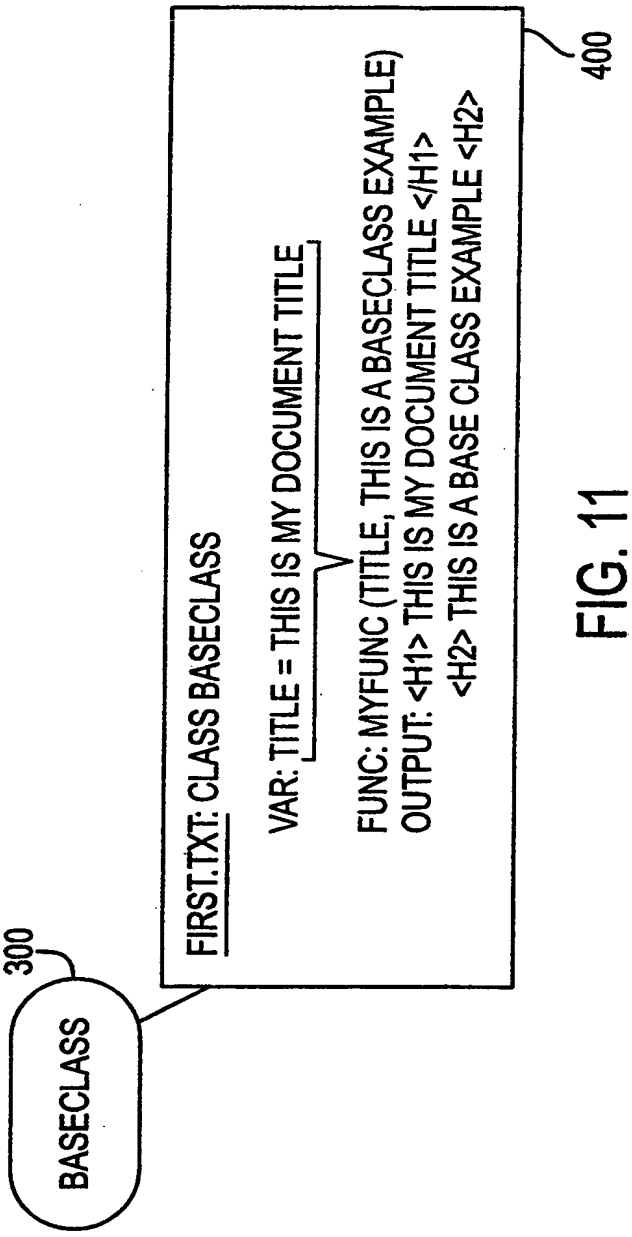

300
BASECLASS
FIRST.TXT: CLASS BASECLASS
VAR: TITLE = THIS IS MY DOCUMENT TITLE
FUNC: MYFUNC (TITLE, THIS IS A BASECLASS EXAMPLE)
OUTPUT: <H1> THIS IS MY DOCUMENT TITLE </H1>
<H2> THIS IS A BASE CLASS EXAMPLE <H2>
400

FIG. 11

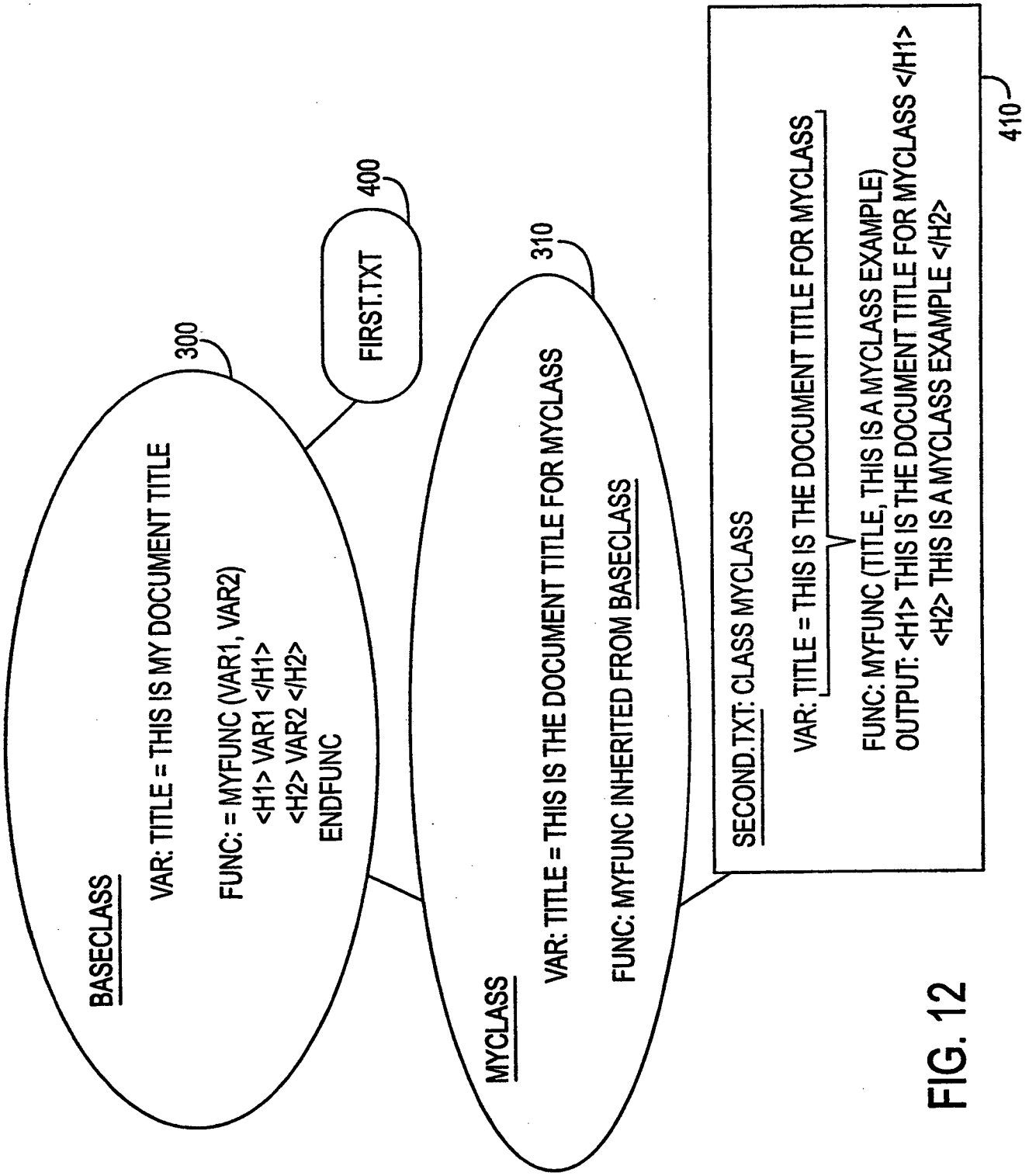

BASECLASS
VAR: TITLE = THIS IS MY DOCUMENT TITLE
FUNC: = MYFUNC (VAR1, VAR2)
<H1> VAR1 </H1>
<H2> VAR2 </H2>
ENDFUNC
300
FIRST.TXT
400
MYCLASS
VAR: TITLE = THIS IS THE DOCUMENT TITLE FOR MYCLASS
FUNC: MYFUNC INHERITED FROM BASECLASS
310
SECOND.TXT: CLASS MYCLASS
VAR: TITLE = THIS IS THE DOCUMENT TITLE FOR MYCLASS
FUNC: MYFUNC (TITLE, THIS IS A MYCLASS EXAMPLE)
OUTPUT: <H1> THIS IS THE DOCUMENT TITLE FOR MYCLASS </H1>
<H2> THIS IS A MYCLASS EXAMPLE </H2>
410

FIG. 12

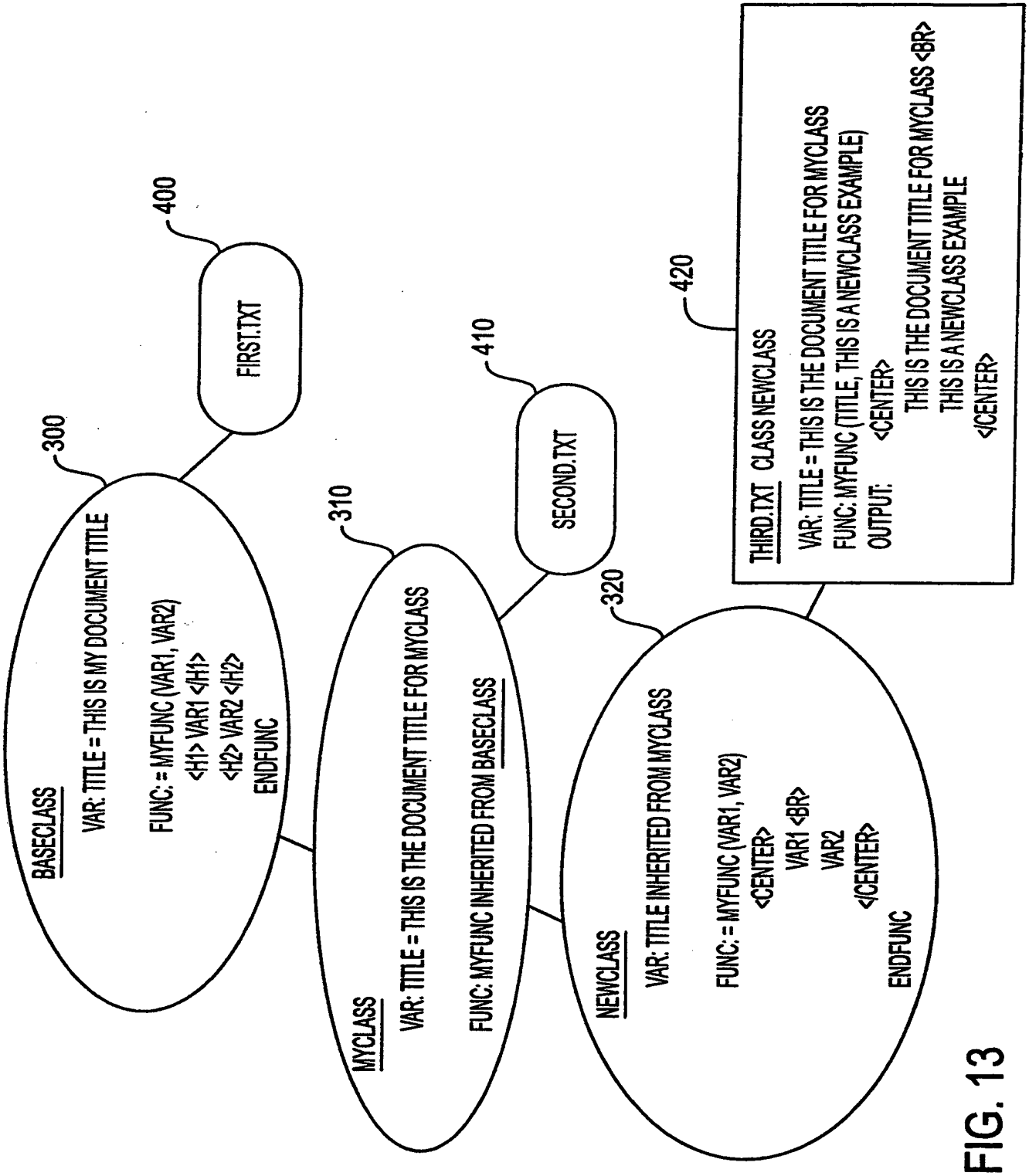

BASECLASS
VAR: TITLE = THIS IS MY DOCUMENT TITLE
FUNC: = MYFUNC (VAR1, VAR2)
<H1> VAR1 </H1>
<H2> VAR2 </H2>
ENDFUNC
300
FIRST.TXT
400
MYCLASS
VAR: TITLE = THIS IS THE DOCUMENT TITLE FOR MYCLASS
FUNC: MYFUNC INHERITED FROM BASECLASS
310
SECOND.TXT
410
NEWCLASS
VAR: TITLE INHERITED FROM MYCLASS
FUNC: = MYFUNC (VAR1, VAR2)
<CENTER>
VAR1
VAR2
</CENTER>
ENDFUNC
320
THIRD.TXT CLASS NEWCLASS
VAR: TITLE = THIS IS THE DOCUMENT TITLE FOR MYCLASS
FUNC: MYFUNC (TITLE, THIS IS A NEWCLASS EXAMPLE)
OUTPUT: <CENTER>
THIS IS THE DOCUMENT TITLE FOR MYCLASS
THIS IS A NEWCLASS EXAMPLE
</CENTER>
420

FIG. 13

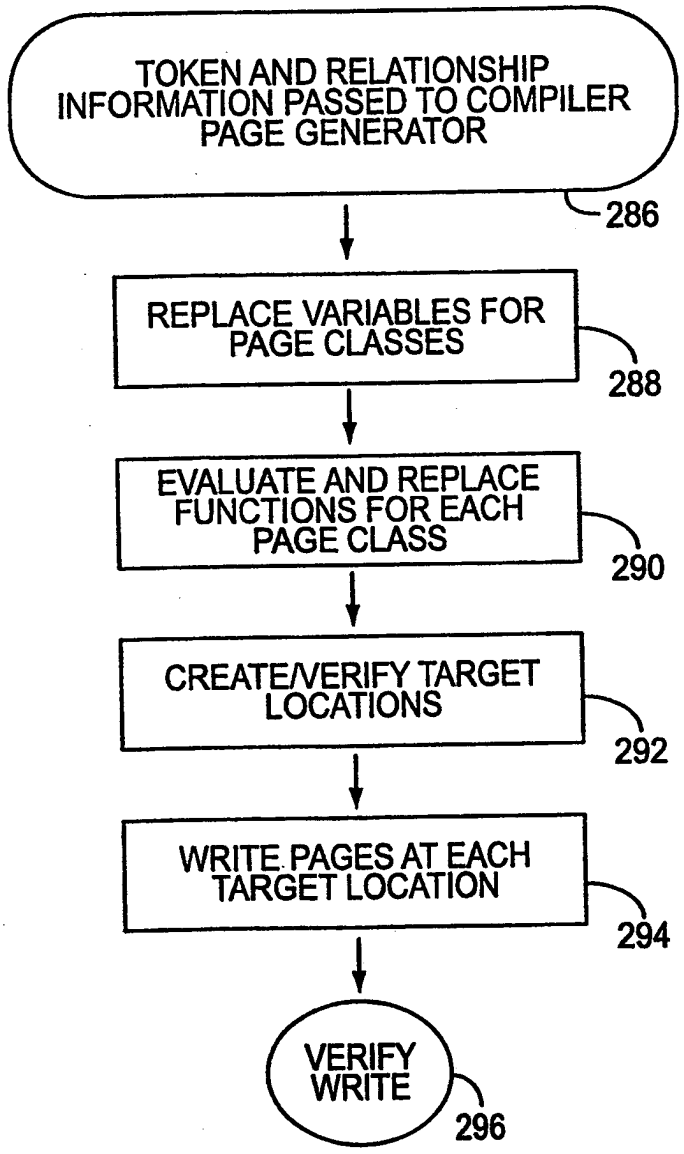

TOKEN AND RELATIONSHIP INFORMATION PASSED TO COMPILER PAGE GENERATOR
286
REPLACE VARIABLES FOR PAGE CLASSES
288
EVALUATE AND REPLACE FUNCTIONS FOR EACH PAGE CLASS
290
CREATE/VERIFY TARGET LOCATIONS
292
WRITE PAGES AT EACH TARGET LOCATION
294
VERIFY WRITE
296

FIG. 14